October 2003

Web agents for requirements consistency management

Z. Chen
University of Wollongong

Aditya K. Ghose
University of Wollongong, aditya@uow.edu.au

Web agents for requirements consistency management

Abstract

Inconsistency handling is an aspect of requirements engineering that has attracted considerable research attention. We explore novel ways to applying semantic Web technologies to this problem, in the context of a Web-based agent-mediated environment for distributed requirements engineering.

Disciplines

Physical Sciences and Mathematics

Publication Details

This article was originally published as: Chen, Z, Ghose, A, Web agents for requirements consistency management, Proceedings of IEEE/WIC International Conference on Web Intelligence (WI 2003), 13-17 October 2003, 710-713. Copyright IEEE 2003.

Web Agents for Requirements Consistency Management

Zerong Chen and Aditya Ghose
Decision Systems Lab, School of IT and Computer Science, University of Wollongong, NSW 2522 Australia
{zrc01,aditya}@uow.edu.au

Abstract

Inconsistency handling is an aspect of requirements engineering that has attracted considerable research attention. This paper explores novel ways to applying semantic web technologies to this problem, in the context of a web-based agent-mediated environment for distributed requirements engineering.

1. Introduction

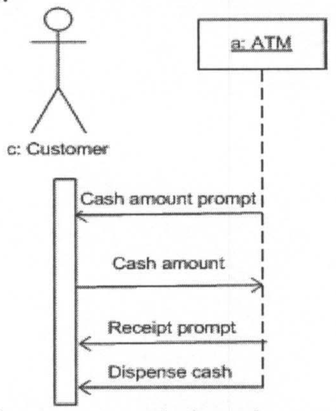
Inconsistency handling is widely acknowledged as a difficult problem in requirements engineering [7, 11]. Our motivations are two-fold. First, we wish to provide *automated support for inconsistency detection and resolution* in the context of industry-standard requirements specification notations. Second, we wish to support *distributed requirements consistency management*, given that most present-day development projects involve distributed sets of stakeholders, often with strict privacy constraints (such as private definitions of requirements consistency, encoded in private sets of business rules). This paper focuses primarily on a solution to the first problem, but does so in the context of, and in a manner synergistic with, a solution to the second. The crux of our proposal is a novel application of semantic web technologies: we propose to use semantic markup of semi-formal and informal requirements specifications to abstract out formal representations that can be used for inconsistency detection and resolution. We describe a preliminary methodology for doing this, in the context of an informal notation (plain text English) and a semi-formal notation (UML sequence diagrams). We suggest how these methodological guidelines can form the basis for end-user semantic markup tools (often referred to as annotators), which, when used in conjunction with notation-specific editors, can form the basis for a practical approach to semantic markup of specifications. This proposal relies on the existence of end-user markup tools that ease the process of semantic markup of specifications and there are reasons to be confident that this is a feasible proposition. Several annotation support tools exist, including AeroDAML [10], COHSE [2], MnM [12], OntoAnnotate [14], OntoMat-Annotizer [14] and the SHOE Knowledge Annotator [9]. We can thus visualize a UML editor such as ArgoUML [1] being augmented with an annotator plug-in which generates a graphical depiction of class and property hierarchies on a side-bar and which permits users to drag-and-drop classes and properties on elements of the UML diagram being edited to automatically generate an underlying markup file. A similar interface can also be visualized for free-form English specifications being edited by any standard text/document editor. We describe a tool called SC-CHECK (Semantic Consistency CHECK) that implements our approach to inconsistency detection. This tool approach has been developed in the context of an agent-mediated architecture for web-based distributed requirements engineering. This constitutes a key element of the REAGENT project [8] that seeks to develop an agent-mediated infrastructure to support requirements inconsistency management, negotiation and model-based monitoring throughout the software life-cycle. The architecture involves two kinds of agents. *Stakeholder agents* pro-actively elicit, manage, negotiate and monitor goals for a single stakeholder. A *requirements repository agent* provides a globally visible store of the requirements that all stakeholders agree to, and executes other functions critical to inconsistency detection and negotiation. The SC-CHECK system functionality may be included in both these kinds of agents, depending on the mode in which REAGENT agents are being used. We omit further details of the REAGENT architecture here for brevity, but point readers to the full version of the paper for details.

2. Semantic markup of specifications

UML class diagrams represent a special case where the structural similarity with ontologies permits easy translation to DAML+OIL. This has led to the development of tools such as DUET [6] that acts as a plug-in to the ArgoUML editor and permits the translation of class diagrams to DAML+OIL. In the following, we shall discuss the markup of English text and UML sequence diagrams. Consider the following fragment of a specification for a banking system: "*A customer can have many accounts. However, an account can belong to exactly one customer. There are two types of accounts: debit account and credit account. All types of accounts should have a balance attribute, and credit accounts should also have a limit attribute to define the* *credit limit. (Rule 2.1.4) If the balance of a customer's credit account is over the limit, an over limit warning operation must be executed automatically. There are two types of staff: tellers and financial officers. (Rule 2.1.3) Only financial officers have the authority to open credit accounts."* Concepts are marked up by referring to pre-defined class definitions in an ontology. Sub-classes can be marked up by referring to their super-classes (this would be relevant in the case of "credit" and "debit" accounts in this example). Attribute names are marked up by using pre-defined data type properties in ontologies (for instance, the "balance" attribute of an account). Operation names are marked up by using pre-defined object properties. Thus, the "over limit warning" operation referred to in the text above is marked up using an object property with the same name that has the "Account" class as its domain and a class called "Operation" as its range. This establishes a link between the "over limit warning" operation and the "Account" class (in the sense that this operation is only invoked by conditions relating to the "Account" class). Using the class "Operation" as the range of this object property identifies this property as an operation. An alternative is to consider operations as concepts/classes in their own right, but this makes it difficult to establish connections between classes that are involved in invoking the operation, or that supply inputs to the operation (this would involve defining a whole new set of object properties). Associations between classes are marked up using pre-defined object properties. The classes participating in the binary association appear in the domain and range respectively. We define a class called "EntityRelation" for this purpose with "hasMaximum", "hasMinimum" and "hasExactly" as integer datatype properties to denote the maximum, minimum and exact cardinalities of the association (the "hasMaximum" property has multiple ranges consisting of both integers and strings to permit us to specify "many" without specifying an integer as the upper limit). Notice that this only defines one "half" of an association, the other "half" also needs to be separately and explicitly defined. Words/phrases which do not fit the above categories are marked up using the *'Words'* tag. We expect that finer-grained categorization of such words can be developed in the future. The following is a portion of the markup file for the text presented above. This approach is based on similar ideas contained in the OntoBroker system[13].

```
<bank:Words>A </bank:Words>
<bank:Customer>customer</bank:Customer>
<bank:Words>can have</bank:Words>
<bank:Customer rdf:about="#customer">
  <hasEntityRelation rdfs:resource="#cust_acc">
</bank:Customer>
<bank:EntityRelation rdf:ID="cust_acc">
<bank:isRelatedTo rdf:resource="#account"/>
<bank:hasMaximum>many</bank: hasMaximum>
</bank:EntityRelation>
<bank:Account>account</bank:Account>
<bank:Words>, however, an</bank:Words>
<bank:Account>account</bank:Account>
<bank:Words> can belong to exactly </bank:Words>
<bank:Account rdf:about="#account">
  <hasEntityRelation rdfs:resource="#acc_cust">
</bank:Account>
<bank:EntityRelation rdf:ID="acc_cust">
<bank:isRelatedTo rdf:resource="#customer"/>
<bank:hasExactly>1</bank:hasExactly>
</bank:EntityRelation>
```

We shall use a sequence diagram (specifying the cash withdrawal process from an ATM) to show how a diagrammatic, semi-formal specification can be marked up.



Objects are marked up by creating instances of pre-defined classes in the ontology. Messages are marked up using the *'Message'* class that we define in the ontology. The *'Message'* class has several properties: The "fromObject" and "toObject" object properties are used for marking up the message directions by linking objects with the "Message" class. The *'hasData'* property (a datatype property) is used for marking up the message itself. The *'sequence'* property is used for marking up the message sequence (denoted by integers).

The code fragment below shows the markup of *'c:Customer'*, *'a:ATM'*, and the first two messages.

```
<bank:Customer rdf:ID="c"/>
<bank:ATM rdf:ID="a"/>
<bank:Message rdf:ID="m1">
<hasData>Cash amount prompt</hasData>
 <fromObject rdfs:resource="#a"/>
 <toObject rdfs:resource="#c"/>
 <sequence>1</sequence>
</bank:Message>
<bank:Message rdf:ID="m2">
 <hasData>Cash amount</hasData>
 <fromObject rdfs:resource="#c"/>
 <toObject rdfs:resource="#a"/>
 <sequence>2</sequence>
</bank:Message>
```

An end-user markup tool for sequence diagrams would prompt the user to select the appropriate class every time an object and message is created. The relevant markup can then be generated automatically. Generating pointers between elements in the diagram and markup in the markup file can also be useful. This enables us to modify a diagram without going through the process of re-generating a new markup file but modifying only those portions that are affected (these bits can be identified by tracing back using the pointers).

3. The SC-CHECK architecture

The SC-CHECK system architecture can be viewed as consisting of three major components: First, it includes set of *editors/annotators* for ontologies, rules and specifications respectively. Second, it includes a *repository* for maintaining the currently operative ontology, the currently applicable set of rules (both business rules and notation-specific consistency rules) and the current, consistent specification. The specification itself can be multi-modal, i.e., written in a combination of informal, semi-formal and formal notations. The specification is represented both in the original notation it was written in, and in a formal representation that is obtained via semantic markup and translation (note that this formal representation is likely to be an abstraction of the original specification). Third, it includes an *inconsistency monitor* that serves to detect inconsistencies in a specification. In our prototype implementation, we use *OilEd* [3] as the SC-CHECK ontology editor and DAML+OIL as the ontology representation language. Consistency rules are written in RuleML [4] and Oryx 2 [5] is used as the rule editor. Two kinds of pre-defined consistency rules are possible: First, *structural rules* for enforcing notation-specific consistency (e.g., two classes cannot have the same name in a class-diagram). Second, application-specific *business rules* (an example is one which states that attribute "balance" in class "Account" must be a real number data type). All these rules use classes in the ontology as their predicates and properties as their arguments. For example, the predicate 'hasAttribute' in a RuleML rule can be used to represent a 'daml:DatatypeProperty'. Thus, the datatype of the "balance" attribute of an "Account" is represented as the following predicate (note that the last argument represents the ID of the stakeholder responsible for the specification): *hasAttribute('Account','balance','real',2).* Since specifications can be multi-modal, we would require notation-specific annotators. The annotator would load the relevant ontologies and display concepts and properties on a side bar. The annotator would interact with the users to prompt for links to appropriate concepts and properties every time a new element of the notation (such as a UML sequence diagram message or a UML state diagram state) is introduced in the editor. When the user finishes editing the specification, the annotator would generate semantic markup code automatically. The inconsistency monitor requires machinery to determine whether a given specification violates a given set of consistency rules. We use Prolog for this purpose (it was also adequately expressive for the kinds of consistency rules we wanted to represent).

Translators: As pointed out above, ontologies, consistency rules and multi-modal specifications are represented in a variety of languages. The translator module thus consists of a variety of individual translation engines which all map to the same underlying formal representation language. In our prototype implementation, consistency rules are written in RuleML. We use XSLT style sheets to translate these rules (available in XML form via the Oryx2 rule editor) to Prolog. The Prolog code below is the translated version of Rule 2.1.2 discussed above:

```
'rule212'(SH)      :-
'hasAttribute'('Account','balance',Type,SH),
'\=='(Type,'double').
```

We provide below an example of translation from a specification notation (UML class diagrams) to Prolog. The translation happens in two stages. First, we use the DUET tool (recall that this is an ArgoUML plug-in that generates DAML+OIL markup for class diagrams) [6] to obtain a representation of the class diagram in DAML+OIL. We then use XSLT style sheets to translate these to Prolog. The DAML markup fragment below represents an "Account" an account class in a class diagram. Note that DUET uses two abstract classes "DUET0" and "DUET3" to define an association between two classes because an association has more than one property (i.e. the name of the class that it relates a given class to, its cardinality and the association name). These therefore cannot be represented using '*daml:ObjectProperty*' which can only define one property. The first abstract class "DUET0" specifies the related class (i.e. "Customer") and the relation name (i.e. "has_cust"). The second abstract class "DUET3" defines the cardinality by using the DAML built-in property "minCardinality".

```
<a:Class rdf:about="bank:#Account">
<a:subClassOf rdf:resource="#DUET0"/>
<a:subClassOf rdf:resource="#DUET3"/>
</a:Class>
<a:Restriction rdf:about="#DUET0">
<a:toClass rdf:resource="bank:#Customer"/>
<a:onProperty rdf:resource="bank:#has_cust"/>
</a:Restriction>
<a:Restriction rdf:about="#DUET3"
 a:minCardinality="1">
<a:onProperty rdf:resource="#has_cust"/>
</a:Restriction>
```

The XSLT style sheet takes the class names as a predicate and instance name as argument. For example,

<a:Class rdf:about="bank:#Account">
is translated to

```
  'class'('Account',2).
```

The translation adds the stakeholder ID as the last argument of every predicate. "a:toClass" is translated to "hasObject" and "onProperty" is translated to "hasObjectProperty". Finally, "cardinality" is translated to "hasObjectNo". Note that different DAML cardinalities are translated to distinct arguments. For instance, this example uses "minCardinality" so the translation is minimum "1" and maximum "many" (the second and third arguments of predicate "hasObjectNo" denotes minimum and maximum cardinality). The following Prolog code is the translation of the above markup. Note that the first predicate simply denotes the existence of a stakeholder with the ID "2".

```
'stakeholder'(2).
'class'('Account',2).
'subClassOf'('CreditAccount','Account',2).
'AbsClass'('DUET0',2).
'hasObjectProperty'('DUET0','has_cust',2).
'hasObject'('has_cust','Customer',2).
'hasObjectNo'('has_cust',1,many,1)
```

Reasoner: Ideally, the *reasoner* must be a formal engine that is able to test the consistency of the conjunction of a multi-modal specification, an ontology (or ontologies) and a set of consistency rules.

We have seen in earlier subsections how consistency rules written in RuleML are translated to Prolog. We have also seen examples of how specifications might be translated to Prolog (in the specific instance of UML class diagrams). A simple Prolog program is used to check inconsistency in SC-CHECK. The program involves multiple definitions of a predicate called "inconsistency", each definition corresponding to a specific consistency rule. If the predicate is satisfied by a given definition, this entails that the corresponding consistency rule has been violated. The violated consistency rule is identified to the user, and, where possible, a message is generated providing additional details on the source of the inconsistency that would provide guidance to the user in appropriately modifying the specification to resolve the inconsistency. The following are some examples of this. Failure to prove the goal "inconsistency" implies that no consistency rule has been violated.

```
inconsistency :- check111(C,A,S),!,
write('Consistency rule 111 fired: '),
write('class: '),write(C),
write(' of stakeholder No.'), write(S),
write(' has duplicated attributes'), write(A),
write('. Please modify.').
```

Feedback Generator: Our focus till now has been on inconsistency *detection*. The key function of the feedback generator module is inconsistency *resolution*. Some indication of how inconsistency resolution might be supported is available in the examples of the definition of the "inconsistency" predicate provided above. These Prolog clauses identify which consistency rule has been violated and also identify, where possible, details of which elements of a specification might need repair, thus providing useful guidance to analysts attempting to resolve such inconsistencies.

4. Conclusions

We have presented a novel approach to automated requirements consistency management using semantic web technology. The deployment of this approach in the context of a web agent-mediated system for distributed requirements engineering appears to be particularly useful. We have tested some of these ideas in a medium-scale case study involving a case-tracking system for a law enforcement agency. The preliminary results of this study appear promising.

5. References

[1] ArgoUML, http://argouml.tigris.org/

[2] S. Bechhofer and C. Goble. Towards Annotation Using DAML+OIL. *Proc. of KCAP'01 Workshop on Semantic Markup and Annotation.* Victoria, Canada.

[3] S. Bechhofer and G. Ng. OILED 3.4 http://oiled.man.ac.uk/

[4] Harold Boley, et. al.. RuleML. http://www.dfki.uni-kl.de/ruleml/

[5] J. B. Dietrich. Oryx 2. http://www.jbdietrich.de/

[6] DUET, CODIP , http://www.ontoprise.de

[7] A. K. Ghose. Formal tools for managing inconsistency and change in RE. *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD 2000)*, San Diego, IEEE Computer Society Press.

[8] A. K. Ghose. Agent-based support for user goals: An outline of the REAGENT framework. Position paper. *Proc. of 2000 Workshop on Agent-Oriented Information Systems (AOIS-2000).* http://www.aois.org.

[9] J. Heflin and J. Hendler. A Portrait of the Semantic Web in Action. *IEEE Intelligent Systems,* 16(2), 2001.

[10] P. Kogut and W. Holmes. AeroDAML: Applying Information Extraction to Generate DAML Annotations from Web Pages. *Proc. of 1st Int'l Conf. on Knowledge Capture (K-CAP 2001).*

[11] A. van Lamsweerde, R. Darimont, and E. Letier, Managing conflicts in goal-driven requirements engineering. *IEEE Trans. on Software Engineering* 24, 11 (1998).

[12] E. Motta, M. Vargas-Vera, J. Domingue, M. Lanzoni and F. Ciravegna. MnM: Ontology Driven Semi-Automatic and Automatic Support for Semantic Mark-up. *Proc. Of ECAI-2002 Workshop on Semantic Authoring, Annotation & Knowledge Markup.*

[13] OntoAnnotate, Ontoprise, http://www.ontoprise.de

[14] S. Staab, A. Maedche and S. Handschuh. An Annotation Framework for the Semantic Web. In *The First International Workshop on MultiMedia Annotation,* Tokyo, Japan, January 2001.